CLAIMS

What is claimed is:

1. A parallel processor program for defining a processor integrated circuit, comprising:

a plurality of processor commands with addresses, said plurality of processor commands comprising a starting processor command, and each of said plurality of processor commands comprising one or more subcommands;

wherein when said processor integrated circuit executes said parallel processor program, said processor integrated circuit executes said staring processor command first and then executes rest of said plurality of processor commands based on an order of said addresses; and

when said processor integrated circuit executes one of said plurality of parallel processor commands, said processor integrated circuit executes all subcommands included in said one of said plurality of processor commands in parallel in one clock cycle.

2. The parallel processor program of claim 1, wherein when said processor integrated circuit executes said parallel processor program, said processor integrated circuit executes said each of said plurality of processor commands in a clock cycle.

3. The parallel processor program of claim 1, wherein said order of said addresses is in ascending order.

4. The parallel processor program of claim 1, wherein said one or more subcommands are chosen from a group consisting of jump-subcommands and non-jump-subcommands.

5. The parallel processor program of claim 4, wherein said non-jump-subcommands comprising:

a subcommand for finishing execution of said parallel processor program;

subcommands for reading and writing unsigned integer variables and arrays; and

subcommands for putting, reading and removing values from a stack, said stack used for keeping temporary results.

6. The parallel processor program of claim 4, wherein said jump-subcommands comprising:

subcommands for calling a function (CALL <address>) and returning back from said function (RETURN);

JUMP <address>;

ZERO_JUMP <variable_name> <address>;

NONZERO_JUMP <variable_name> <address>;

LOOP_INC_NOLESS <variable_name> <constant_value> <address>;

LOOP_INC_NOMORE <variable_name> <constant_value> <address>;

LOOP_DEC_NOLESS <variable_name> <constant_value> <address>; and

LOOP_DEC_NOMORE <variable_name> <constant_value> <address>.

7. A method for translating a C++ program into a parallel processor program with a minimum size defining a processor integrated circuit, said parallel processor program written in a parallel processor language, comprising:

(a) dividing said C++ program into plurality of C++ functions allowed to call each other;

(b) translating said plurality of C++ functions into a plurality of blocks TRANS(<C++ function>) written in said parallel processor language; and

(c) concatenating said plurality of blocks TRANS(<C++ function>) into said parallel processor program.

8. The method of claim 7, wherein said step (b) comprising presenting one of said plurality of blocks TRANS(<C++ function>) as follows:

0: TRANS(<C++ function body>)
1: <function_finish_command>

wherein said TRANS(<C++ function body>) is a block obtained as a result of translating a C++ command <C++ function body> into said parallel processor language, and said *<function_finish_command>* is a processor command as follows:

(1) when said <C++ function> is a main function, then said command *<function_finish_command>* is command: *FIN*;

(2) when said <C++ function> is any void function except said main function, then said command *<function_finish_command>* is command: *RETURN*; and

(3) when said <C++ function> is any unsigned integer function, then said command *<function_finish_command>* is empty.

9. The method of claim 7, wherein said step (b) comprising translating a return-operator as follows:

(1) when said return-operator is inside a void function: *return*, creating TRANS(<return-operator>) as a command: *RETURN*.

(2) when said return-operator is inside a function that returns an unsigned integer value: *return* *<expression>*, creating TRANS(<return-operator>) as follows:

0: TRANS(<expression>)
1: <put_value_command>

wherein said command <put_value_command> includes 2 subcommands: <put_value_subcommand1> and said command: *RETURN*, and said subcommand <put_value_subcommand1> is created according to following rules:

a) when said expression <expression> is a constant, then said subcommand <put_value_subcommand1> is:

PUT_CONST <expression>;

b) when said expression <expression> is not a constant and EXPR_VAR(<expression>)==STCK_0, then said subcommand <put_value_subcommand1> is empty; and

c) when said expression <expression> is not a constant and EXPR_VAR(<expression>)!=STCK_0, then said subcommand <put_value_subcommand1> is:

PUT EXPR_VAR(<expression>).

10. The method of claim 7, wherein said step (b) comprising translating a set-operator as follows:

(1) when *<variable_name>* = *<expression>*, creating TRANS(<set-operator>) as:

0: TRANS(<expression>)

1: <set_command>

wherein said command <set_command> is created according to following rules:

a) when said expression <expression> is a constant, then said command <set_command> is:

SET_CONST <variable_name> <expression>;

b) when said expression <expression> is not a constant and EXPR_VAR(<expression>)==STCK_0, then said command <set_command> is:

SET <variable_name> STCK_0; DROP 1;

and

c) when said expression <expression> is not a constant and EXPR_VAR(<expression>)!=STCK_0, then said subcommand <set_command> is:

SET <variable_name> EXPR_VAR(<expression>);

and

(2) when *<unsigned_integer_array>[<index_expression>]* = *<expression>*, creating TRANS(<set-operator>) as:

0: TRANS(<expression>)

1: <save_expression_command>

2: TRANS(<index_expression>)

3: <set_command>

wherein said command <save_expression_command> is obtained according to following rules:

a) when said TRANS(<expression>) is empty, said block TRANS(<index_expression>) is empty, or EXPR_VAR(<expression>)==STCK_0, then said command <save_expression_command> is empty;

b) otherwise, said command <save_expression_command> is:

PUT EXPR_VAR(<expression>);

and said command <set_command> includes 2 subcommands: <set_subcommand1> and <set_subcommand2> obtained according to following rules:

a) when said expression <expression> is a constant, following cases aa), ab), and ac) are considered:

aa) when said expression <index_expression> is a constant then following are assigned:

<set_subcommand1>: *WRITE_CONST_CONST <unsigned_integer_array> <expression> <index_expression>;*

<set_subcommand2>: empty;

ab) when said expression <index_expression> is not a constant and EXPR_VAR(<index_expression>)==STCK_0, then following are assigned:

<set_subcommand1>: *WRITE_CONST_VAR <unsigned_integer_array> <expression> STCK_0;*

<set_subcommand2>: *DROP 1*;

ac) when said expression <index_expression> is not a constant and EXPR_VAR(<index_expression>)!=STCK_0, then following are assigned:

<set_subcommand1>: *WRITE_CONST_VAR <unsigned_integer_array> <expression> EXPR_VAR(<index_expression>);*

<set_subcommand2>: empty;

b) when said expression <expression> is not a constant and at least one of the following two conditions is valid:

- EXPR_VAR(<expression>)==STCK_0; and
- said command <save_expression_command> is not empty,

then following cases ba), bb), and bc) are considered:

ba) when said expression <index_expression> is a constant then following are assigned:

<set_subcommand1>: *WRITE_VAR_CONST <unsigned_integer_array> STCK_0 <index_expression>;*

<set_subcommand2>: *DROP 1*;

bb) when said expression <index_expression> is not a constant and EXPR_VAR(<index_expression>)==STCK_0, then following are assigned:

<set_subcommand1>: *WRITE_VAR_VAR <unsigned_integer_array> STCK_1 STCK_0;*

<set_subcommand2>: *DROP 2*.

bc) when said expression <index_expression> is not a constant and EXPR_VAR(<index_expression>)!=STCK_0, then following are assigned:

<set_subcommand1>: *WRITE_VAR_VAR <unsigned_integer_array> STCK_0 EXPR_VAR(<index_expression>);*

<set_subcommand2>: *DROP 1;*

c) when said expression <expression> is not a constant, EXPR_VAR(<expression>)!=STCK_0, and said command <save_expression_command> is empty, then following cases ca), cb), and cc) are considered:

ca) when said expression <index_expression> is a constant, then following are assigned:

<set_subcommand1>: *WRITE_VAR_CONST <unsigned_integer_array> EXPR_VAR(<expression>) <index_expression>;*

<set_subcommand2> empty.

cb) when said expression <index_expression> is not a constant and EXPR_VAR(<index_expression>)==STCK_0, then following are assigned:

<set_subcommand1>: *WRITE_VAR_VAR <unsigned_integer_array> EXPR_VAR(<expression>) STCK_0;*

<set_subcommand2>: *DROP 1.*

cc) if the expression <index_expression> is not a constant and EXPR_VAR(<index_expression>)!=STCK_0, then following are assigned:

<set_subcommand1>: *WRITE_VAR_VAR <unsigned_integer_array> EXPR_VAR(<expression>) EXPR_VAR(<index_expression>);*

<set_subcommand2>: empty.

11. The method of claim 7, wherein said step (b) comprising translating a call-operator as follows:

defining TRANS(<call-operator>) as:

TRANS(<call-operator expression>).

12. The method of claim 7, wherein said step (b) comprising translating an if-operator as follows:

when said if-operator is presented as follows:

if (<expression>) <positive_operator>
else <negative_operator>

wherein said operators <positive_operator> and <negative_operator> are allowed to be empty,

creating TRANS(<if-operator>) as follows:

a) when said expression <expression> is a constant equal to zero, then said TRANS(<if-operator>) is TRANS(<negative_operator>);

b) when said expression <expression> is a constant that is not equal to zero, then said TRANS(<if-operator>) is TRANS(<positive_operator>);

c) when said expression <expression> is not a constant and both TRANS(<negative_operator>) and TRANS(<positive_operator>) are empty blocks, then said TRANS(<if-operator>) is empty;

d) when said expression <expression> is not a constant, TRANS(<negative_operator>) is an empty block, and

TRANS(<positive_operator>) is not an empty block, then said TRANS(<if-operator>) is:

0: TRANS(<expression>)

1: <compare_jump_command>

2: TRANS(<positive_operator>)

wherein said command <compare_jump_command> includes a first subcommand and a second subcommand: when EXPR_VAR(<expression>)==STCK_0, then said first subcommand is *DROP 1*, otherwise said first subcommand is empty; and said second subcommand is:

ZERO_JUMP EXPR_VAR(<expression>) <address>

wherein said <address> is an address of said first command that situates after said TRANS(<positive_operator>).

e) when said expression <expression> is not a constant, TRANS(<negative_operator>) is not an empty block, and TRANS(<positive_operator>) is an empty block, then said TRANS(<if-operator>) is:

0: TRANS(<expression>)

1: <compare_jump_command>

2: TRANS(<negative_operator>)

wherein said command <compare_jump_command> includes a third subcommand and a fourth subcommand: when EXPR_VAR(<expression>)==STCK_0, then said third subcommand is *DROP 1*, otherwise said thid subcommand is empty; and said fourth subcommand is:

NONZERO_JUMP EXPR_VAR(<expression>) <address>

wherein said <address> is an address of said first command that situates after said TRANS(<negative_operator>).

f) when said expression <expression> is not a constant and both TRANS(<negative_operator>) and TRANS(<positive_operator>) are not empty blocks, then said TRANS(<if-operator>) is;

0: TRANS(<expression>)

1: <compare_jump_command>

2: TRANS(<positive_operator>)

3: JUMP <address of the first command after the block TRANS(negative_operator)>

4: TRANS(<negative_operator>)

wherein said command <compare_jump_command> includes a fifty subcommand and sixth subcommand: when EXPR_VAR(<expression>)==STCK_0, then said fifth subcommand is *DROP 1*, otherwise said fifth subcommand is empty; and said sisth subcommand is:

ZERO_JUMP EXPR_VAR(<expression>) <address>

wherein said <address> is an address of said first command of said TRANS(<negative_operator>).

13. The method of claim 7, wherein said step (b) comprising translating a for-operator as follows:

when said for-operator is presented as follows:

for (<at_start>;<expression>;<at_iteration_finish>) <body>

wherein said <at_start> and <at_iteration_finish> are set-operators or empty, and said <body> is any allowed C++ operator,

creating TRANS(<for-operator>) as follows:

a) when said expression <expression> is a constant equal to zero, then said TRANS(<for-operator>) is empty;

b) when said expression <expression> is a constant that is not equal to zero, then said TRANS(<for-operator>) is:

0: TRANS(<at_start>);

1: BODY_BLOCK;

where said BODY_BLOCK is:

0: TRANS(<body>);

1: TRANS(<at_iteration_finish>);

2: JUMP <address of first command of block BODY_BLOCK>

c) when said expression <expression> is not a constant, then said TRANS(<for-operator>) is:

0: TRANS(<at_start>);

1: TRANS(<expression>);

2: <check_condition_command>

3: TRANS(<body>);

4: TRANS(<at_iteration_finish>)

5: JUMP <address of first command of block TRANS(<expression>)+<check_condition_command>>

where said command <check_condition_command> includes a first subcommand and a second subcommand: when EXPR_VAR(<expression>)==STCK_0, then said first subcommand is *DROP 1*, otherwise said first subcommand is empty; and said second subcommand is:

ZERO_JUMP EXPR_VAR(<expression>) <address>

wherein said <address> is an address of said first command that situates after said TRANS(<for-operator>).

14. The method of claim 7, wherein said step (b) comprising translating a begin-end-operator as a concatenation of blocks created for each operator included inside said begin-end-operator.

15. The method of claim 7, wherein said step (b) comprising translating a break-operator into a block TRANS(<break-operator>), said block TRANS(<break-operator>) including a processor command: *JUMP <address of first command after the cycle>*.

16. A computer-readable medium having computer-executable instructions for performing a method for translating a C++ program into a parallel processor program with a minimum size defining a processor integrated circuit, said parallel processor program written in a parallel processor language, said method comprising:

(a) dividing said C++ program into a plurality of C++ functions allowed to call each other;

(b) translating said plurality of C++ functions into a plurality of blocks TRANS(<C++ function>) written in said parallel processor language; and

(c) concatenating said plurality of blocks TRANS(<C++ function>) into said parallel processor program.

17. A method for optimizing execution time of a parallel processor program, comprising:

(a) receiving said parallel processor program;

(b) performing dummy jumps optimization;

(c) performing linear code optimization;

(d) performing jumps optimization;

(e) returning back to said step (b) when there is any change made in said steps (b), (c), and (d); and

(f) outputting a resulted new processor program when there is no change made in said steps (b), (c), and (d).

18. The method of claim 17, wherein said step (b) comprising:

(b1) applying transition of jumps;

(b2) removing unreachable jumps; and

(b3) removing dummy jumps.

19. The method of claim 17, wherein said step (c) comprising:

(c1) examining all commands included in a domain <domain>; and

(c2) removing empty commands from said domain <domain>.

20. The method of claim 19, wherein said step (c1) comprising performing following steps for a subcommand <subcommand> included in a command with an address K:

a) evaluating sets LOCK(<subcommand>) and ACCESS(<subcommand>);

b) finding such a maximal address T_LOCK that K>T_LOCK>=ADDR and a command with said address T_LOCK includes a subcommand <subcommand2> so that set

LOCK(<subcommand2>) intersects with said set LOCK(<subcommand>), and assigning T_LOCK=-1 when there is no said address T_LOCK;

c) finding such a maximal address T_LOCK_ACCESS that K>T_LOCK_ACCESS>=ADDR and a command with said address T_LOCK_ACCESS includes a subcommand <subcommand3> so that set ACCESS(<subcommand3>) intersects with said set LOCK(<subcommand>), and assigning T_LOCK_ACCESS=-1 when there is no said address T_LOCK_ACCESS;

d) finding such a maximal address T_ACCESS_LOCK that K>T_ACCESS_LOCK>=ADDR and a command with said address T_ACCESS_LOCK includes a subcommand <subcommand4> so that set LOCK(<subcommand4>) intersects with said set ACCESS(<subcommand>), and assigning T_ACCESS_LOCK=-1 when there is no said address T_ACCESS_LOCK;

e) defining an address T = max (ADDR, T_LOCK+1, T_LOCK_ACCESS, T_ACCESS_LOCK+1), where said address T may take values K>=T>=ADDR;

f) going to step i) below when T==K, otherwise going to step g) below;

g) when SIZE(command with said address T)<N, appending said subcommand <subcommand> to a command with said address T, removing said subcommand <subcommand> from said command with said address K, and going to step i) below.

h) setting T=T+1, and going back to said step f) above; and

i) finishing processing said subcommand <subcommand>.

21. The method of claim 17, wherein said step (d) comprising performing following steps for a command <command>, said command <command> having a previous command immediately preceding said command <command>, said steps comprising:

(1) when said previous command has no jump-subcommand, said command <command> has at least one jump-subcommand and does not depend from said previous command, and SIZE(said previous command)+SIZE(said <command>) <= N, performing following steps a) and b):

a) when said command <command> is not a jump-target command, replacing said previous command with a union of said previous command and said command <command>, and removing said command <command>;

b) when said command <command> is a jump-target command, and said command <command> contains jump-subcommand JUMP or jump-subcommand RETURN, replacing said previous command with a union of said previous command and said command <command>;

(2) when said previous command has at least one jump-subcommand but has no jump-subcommands of types CALL, RETURN and JUMP (consequently, it has jump-subcommands of types LOOP*, ZERO_JUMP and NONZERO_JUMP only), and said command <command> does not depend from said previous command, performing following steps a), b) and c):

a) when said command <command> is not a jump-target command, and SIZE(said previous command)+SIZE(said <command>) <= N, replacing said previous command with a union of said previous command and said command <command>, and removing said command <command>;

b) when said command <command> is a jump-target command and contains jump-subcommand JUMP or jump-

subcommand RETURN, and SIZE(said previous command)+SIZE(said <command>) <= N, replacing said previous command with a union of said previous command and said command <command>;

c) when said command <command> is a jump-target command, said step b) of said step (2) is not applicable, and SIZE(said previous command)+SIZE(said <command>) < N, replacing said previous command with a union of said previous command and said command <command>, and appending a subcommand JUMP <ADDR+1> to an end of said newly replaced previous command, where ADDR is an address of said command <command>;

(3) when said command <command> has a jump-subcommand <subcommand> of type JUMP or CALL, a jump-target command of said subcommand <subcommand> has no jump-subcommand and does not depend from said command <command>, and SIZE(said <command>)+SIZE(said jump-target command)<=N, inserting all subcommands of said jump-target command to said command <command> before said jump-subcommand <subcommand>, and increasing a jump-address in said subcommand <subcommand> by 1;

and

(4) when said command <command> has a jump-subcommand <subcommand> of type JUMP, and a jump-target command of said subcommand <subcommand> has at least one jump-subcommand and does not depend from said command <command>, performing following steps a), and b):

a) when said jump-target command has only one jump-subcommand <subcommand2> of types JUMP or CALL and has no more jump-subcommands (and it may have at least one non-

jump-subcommand), and SIZE(said <command>)+SIZE(said jump-target command)<=(N+1), inserting all subcommands of said jump-target command into said command <command> before said jump-subcommand <subcommand>, and removing said jump-subcommand <subcommand> from said command <command>;

b) when said jump-target command has at least one non-jump-subcommand of any types and at least one jump-subcommand of types LOOP*, ZERO_JUMP and NONZERO_JUMP only, and SIZE(said <command>)+SIZE(said jump-target command)<=N, inserting all subcommands of said jump-target command into said command <command> before said jump-subcommand <subcommand>, and replacing said jump-subcommand <subcommand> with a subcommand JUMP <ADDR+1>, where said ADDR is an address of said jump-target command.

22. A computer-readable medium having computer-executable instructions for performing a method for optimizing execution time of a parallel processor program, said method comprising:

(a) receiving said parallel processor program;

(b) performing dummy jumps optimization;

(c) performing linear code optimization;

(d) performing jumps optimization;

(e) returning back to said step (b) when there is any change made in said steps (b), (c), and (d); and

(f) outputting a resulted new processor program when there is no change made in said steps (b), (c), and (d).